Reportable Rest Services Interfaces

30 July 2009

Version 0.3

Bill Jones / Tack Tong

Revision History

Date	Version	Description	Author
2/26/2009	0.1	Initial Version	Bill Jones
3/19/2009	0.1	Incorporated feedback from Tack and flushed out more details. Show markup to see the changes.	Bill Jones
4/13/2009	0.2	Initial draft ready for review.	Bill Jones
5/26/2009	0.3	Prepared draft for review	Bill Jones

Table of Contents

1 Introduction

REST services are a class of web applications deployed on Web servers. REST stands for Representational State Transfer, meaning that a REST service returns a representation of resources stored on the server. Although REST services can represent resources using any data format, this specification focuses on REST using XML. XML data from a REST service can be streamed to a reporting client as the data source for report generation. However, there are some issues that may occur when attempting to create production systems using REST as an XML source for reporting. This document defines the requirements for a new class of REST service called a Reportable REST Service. Reportable REST Services implement features that are designed to work with data warehousing, reporting and document generation tools.

> *Data Warehousing* is the process of extracting data from operational systems for storage in a data warehouse. An operational system is an application being used by individuals to do work. This includes creating and editing objects that are later extracted from the system and stored in the data warehouse. Data warehousing typically extracts all objects, or some well defined sub-set of objects stored in the operational system. For this reason, it is important for Reportable REST Services for operational systems containing large data volumes to support the enterprise scalability requirements.
>
> *Reporting* is the collection of tabular data used to create a tabular report or chart. The content of each table or chart in the report is the result of one REST GET from the REST service. Reports may contain multiple tables, but each table is a different REST GET. Typically reports use a filter to select a subset of the data available from the reportable REST service.
>
> *Document Generation* is the collection of data used to create documents. Documents typically have a hierarchical organization compared to the tabular organization of reports. This hierarchy maps well into an XML document structure. The generation of a document may use one or more REST GETs to collect the content. The parent child containment of an XML document may be used to create section/subsection organization in the generated document. Each subsection may be generated based on child elements in an XML document, or may be based on the results of a separate REST GET. Generated documents can vary from small filtered sub-sets to very large listings of all objects in the operational system.

A REST service that implements any of the Reportable REST services requirements is a ***Data Service.***

2 Reportable REST Services Requirements

This section outlines the Reportable REST Services requirements. Each requirement enables different functional capabilities in the service and is derived from general reporting and data warehousing solutions. The purpose of these requirements is to implement services that can be used by the data warehousing, reporting or document generation tools to integrate with any product without making source code changes to the tool. They are designed to allow the configuration of these tools without special knowledge of how the individual REST services work. Common user interface features are built on these capabilities, further simplifying the administration tasks. In other words, administrators are able to focus on the business objects presented by Reportable REST services without needing to know how to use the REST service to get data.

#	Requirement	Required?

General		
3.1	*Self-Contained XML:* The data returned for a resource must be self-contained in the xml document	Mandatory
3.2	*XML Schema:* An xsd schema is required for bulk data return.	Highly Recommended
3.3	Resource navigation - gradual discovery of resource url (usability feature for long and complex resource url)	Recommended
Enterprise Scalability		
4.1	Paging for large dataset	Recommended for large data sets
4.2	Support delta ETL load - only extract modified data since last ETL	Recommended for large data sets
4.3	Field selection capability to limit the volume of bulk data transferred across the network.	Recommended for complex resource models
4.4	Filtering capability to select specific resources for data generation.	Recommended for large data sets

Figure 1. Reportable REST Services Requirements

Many of these requirements are defined using certain URL arguments, which are described in sections 3 and 4. Although it is not required for Reportable REST services to implement all of these requirements, it is important for them to ignore these parameters without returning errors. The reason is that the requirements do not define a mechanism to enumerate which requirements are supported. This was done to keep the functional requirements as simple as possible. Reportable REST clients can always include the URL arguments regardless of whether a particular service understands what the arguments mean.

Argument	**Requirements**
metadata=schema	3.2
ModifiedSince=*date*	4.2
fields=*xpath expression*	4.3, 4.4

Figure 2 - Mandatory arguments to be accepted by every reportable REST service

The arguments in Figure 2 must be accepted without errors by a reportable REST service. The arguments do not have to be supported, but they must not cause errors when reporting and data warehousing solutions call the service.

3 General Requirements

3.1 Self-Contained XML: *The data returned for a resource must be self-contained in the xml document*

REST URLs return an XML document describing the resource that the URL references. It is common for REST services to return a mixture of XML data and URL references to other resources. In order for data to be self-contained, it must be returned by the first URL request. This is an example of a self contained XML document. This following URL returns a list of defects.

```
http:// 10.0.0.1:9080/RESTServiceName/Defects
```

This is the XML data it returns

```
<DefectList>
    <Defect>
        <id>DEFECT01</id>
        <Headline>spelling error in login screen</Headline>
        <State>Opened</State>
        <Severity>3-Average</Severity>
    </Defect>
    <Defect>
        <id>DEFECT02</id>
        <Headline>sales tax incorrect if item deleted from
            purchase</Headline>
        <State>Resolved</State>
        <Severity>1-Critical</Severity>
    </Defect>
    <Defect>
        <id>DEFECT03</id>
        <Headline>cancel sale doesn't correctly repaint
            screen</Headline>
        <State>Resolved</State>
        <Severity>3-Average</Severity>
    </Defect>
</DefectList>
```

Figure 3 - Self-contained XML document

This document is self contained because the required data, the defect's id, Headline, State and Severity, are all returned by the URL.

This is an example of an REST URL that does not return self-contained data:

```
<DefectList>
    <Defect href=" http://
        10.0.0.1:9080/RESTServiceName/Defects/Defect?id=DEFECT01"
        />
    <Defect href=" http://
        10.0.0.1:9080/RESTServiceName/Defects/Defect?id=DEFECT02"
        />
    <Defect href=" http://
        10.0.0.1:9080/RESTServiceName/Defects/Defect?id=DEFECT03"
        />
</DefectList>
```

Figure 4 - XML document with external references

Each of the URLs in the document would return the id, Headline, State and Severity. The data is not self-contained because it would be necessary for the report generation and metrics collection tools to issue these additional URLs to get all the data.

In theory, a data collection tool could traverse the references in order to collect all the information from the REST service. However, in practice this causes data collection efficiency issues. When collecting large volumes of data, doing http gets on large numbers of URLs would be very inefficient. There is overhead created by each HTTP GET for any application server. Finding the resource referenced in the URL will take some amount of time, which can vary depending on the implementation of the application server. It is much more efficient to return the data in one XML stream rather than processing many.

For this reason, a Reportable REST service must return all required data in a self contained XML document. Note that it is a good practice to include URL references in the data in addition to the required data. For example, the XML shown in Figure 3 could include the href attributes shown in Figure 4. In would still be self-contained because the required data is included in the document. Therefore it is not necessary to HTTP GET the URL to each defect in order to generate the report or collect the metrics.

There is one exception to this rule. This is the paging for large dataset requirement. When data is paged, each page conforms to the same XML schema. The data from all pages could be merged into one XML document and the document would still validate against the schema. This is not true of the general case, where one document contains links to others that have different schemas. The details of how paging works is discussed in section 4.1 Paging for large dataset.

3.2 XML Schema: *An xsd schema is required for bulk data return.*

Any resource may support a metadata description of itself. Although it is not required, it is strongly recommended that each REST URL support returning an XML Schema for its XML Resource format. The schema is returned by adding the “metadata=schema” URL argument to any URL.

The schema is a description of the data available for inclusion in reports or bulk data collections. This may be different than the schema that describes the data returned by the URL because support for different requirements can cause subsets of the XML data to be excluded from the XML data returned. For example, if the field selection capability (see section 4.3) is supported by the REST service, the data returned by the service is a subset of the data described by the XML schema. It is important not to confuse the two schemas: one that describes all available data and one that describes the data returned using specific URL argument values. This is a subtle but important distinction. Since the only schema needed for reporting and metrics collection is the metadata schema, it is the only one defined by the Reportable REST Services requirements. REST services are free to implement URL arguments that return the schema that describes the data returned using a specific set of arguments.

Here's an example of why the distinction between the metadata schema and regular schema are important. Let's assume that a particular tool defines a Change Request record has one and only one owner. The REST service returns the owner as a child XML element of the Change Request. This child element contains multiple properties of the owner and XML attributes. For reporting purposes, there is a difference between a schema that states a Change Request always has 1 and only 1 owner. If the reporting tool is guaranteed that a particular value will always be defined, the report author's task is simplified because there is no need to define how to handle cases where there is no owner. This will allow the report author to define a filter based on attributes of the owner without having to consider cases where the owner does not exist.

The alternative is to modify the REST service to always list the lower bound cardinality of 0 for the owner, just as it would do for every element in the schema. This would destroy the lower bound information for all resources available for reporting systems.

The following example shows the xml schema for a URL that returns a resource representing a Project. Note that there is no requirement for other REST services returning projects to conform to this schema.

```
<xs:schema elementFormDefault="qualified"
    targetNamespace="http://www.ibm.com/rational/Example/Project">
    <xs:element type="Project" name="Project"/>
    <xs:complexType name=" Project ">
        <xs:all>
            <xs:element minOccurs="1" maxOccurs="1" name="Tasks">
                <xs:complexType>
                    <xs:sequence>
                        <xs:element minOccurs="0" type="Task"
                            maxOccurs="unbounded" name="Task">
                            [Details omitted]
                        </xs:element>
                    </xs:sequence>
                </xs:complexType>
            </xs:element>
            <xs:element minOccurs="1" maxOccurs="1"
                name="Resources">
                <xs:complexType>
                    <xs:sequence>
                        <xs:element minOccurs="0" type="Resource"
                            maxOccurs="unbounded" name="Resource"/>
                    </xs:sequence>
                </xs:complexType>
            </xs:element>
            <xs:element type="xs:string" name="Name">
                [Details omitted]
            </xs:element>
        </xs:all>
    </xs:complexType>
     [Additional definitions of complex types omitted for brevity]
</xs:schema>
```

The schema defines the root Project element and the complex type for the Project. The Project type may contain lists of tasks and resources. This in turn causes the definition of the Task and Resource complex types, which is not shown. This causes the inclusion of the other complex types referenced by those resource types.

The minOccurs and maxOccurs values of 0 and unbounded for the list of tasks and resources reflect the unlimited number of resources each element may contain. They can also be bounded using other values. These values are defined by the REST service generating the schema.

It is critically important that the REST services generate schemas that correctly described the XML returned by the URL. Reporting and data warehousing solutions could generate errors if expected data is not present.

3.3 Resource navigation - gradual discovery of resource url (usability feature for long and complex resource url)

Resource discoverability allows Data Warehousing and reporting solutions to browse the resources supported by a product. Discoverability starts with the root REST Service URL, or with the URL of a known resource. Doing an HTTP GET on the URL returns the xml representation of

that resource. This representation may include the properties of the resource and/or zero or more child resources. Any of the child resources may include their own URL. Examples of resources found during discovery are:

> *ClearQuest query:* This is a resource that returns a set of records. It may either be a user defined query or a pre-defined query returning all records of a given type.
>
> *RequisitePro project:* This is the top level RequisitePro resource. It can be used to get a list of requirements, packages, users, groups or views, as defined by RequisitePro.
>
> *ClearCase vob:* This resource contains resources that represent the files and versions stored in the vob.

Resource discoverability allows the URL for a specific resource to be found programmatically rather than typed in by hand, which requires extensive knowledge of the meaning of each resource identifier in the URL, knowledge of what parameters are required and what values to provide. Resource discoverability is analogous to navigating to a specific page on a web site by following links from the home page and clicking on hyperlinks.

3.3.1 Example: Using a ClearQuest Query Folder to discover a Query

This example shows how a URL to a ClearQuest query folder returns URLs to the queries and query folders it contains. This is the URL to the Public Queries folder in the SAMPL Enterprise database, deployed on the local host.

```
http://localhost:9080/DataServices/ClearQuest/Enterprise/SAMPL/Public+Queries
```

The XML data returned by the REST service includes QueryFolder and Query resources with href attributes. These attributes can be used to return either the contents of another query folder, or the results of a specific query.

```
<QueryFolder Version="1.0.0">
   <DisplayName>Public Queries</DisplayName>
      <QueryFolders>
         <QueryFolder
            href="http://localhost:9080/DataServices/ClearQuest/E
            nterprise/SAMPL/Public+Queries/TM+Queries?Type=Query+
            Folder"/>
         <QueryFolder
            href="http://localhost:9080/DataServices/ClearQuest/E
            nterprise/SAMPL/Public+Queries/TM+Reports?Type=Query+
            Folder"/>
      </QueryFolders>
   <Queries>
      <Query
         href="http://localhost:9080/DataServices/ClearQuest/Ente
         rprise/SAMPL/Public+Queries/All+Defects?Type=Query"/>
      <Query
         href="http://localhost:9080/DataServices/ClearQuest/Ente
         rprise/SAMPL/Public+Queries/Keyword+Search?Type=Query"/>
   </Queries>
</QueryFolder>
```

Figure 5 - ClearQuest Query Folder

In this case, the caller is looking for the URL to the All Defects query, contained in the Public Queries folder. Reporting and data warehousing solutions are able to use this information when users are browsing for URLs to use as the basis of reports or bulk data transfers.

4 Enterprise Scalability Requirements

4.1 Paging for large dataset

A reportable REST service may support returning data across multiple pages. The REST service determines the size of each page based on its internal implementation capabilities and performance issues. When there are additional pages, attributes are added to the root XML element describing how to get the next page of data. There are four XML element attributes defined for paging.

Root element attribute name	Required	Description
href	Yes	The URL to the next page. The format of this URL and its arguments are internal to the REST services and opaque to the caller.
rel	Yes	The value is always "next'. This was added to be compatible with Atom paging.
TotalPages	No	The total number of pages in the set. This value should only be returned if the REST service is able to efficiently compute it. Otherwise it should be omitted.
Page	No	The current page number. This value is only useful to the caller if used in conjuction with TotalPages. It should be omitted if TotalPages is not supported.

Figure 6. Root XML element paging attributes

The overall logic used for Paging is very simple. If the REST service determines that the data requested by a URL needs to be broken up into multiple pages, it will set the attributes of the root XML element as shown in Figure 6. Any element with a sequence of zero or more child elements in the document may be split into multiple pages. Callers check the root element for the paging attributes and continue to HTTP GET the next page until there is no href attribute returned. The content returned across all pages is defined to be the union of all elements in each page.

Note that there is no requirement about the number of pages, the number of records per page, data volume of each page, or the relative size of one page compared to the other. Whether or not data is returned in multiple pages and what logic determine what is on each page is completely up to the implementation of the REST service.

At first paging may seem to be at odds with requirement 3.1, *Self-Contained XML:* The data returned for a resource must be self-contained in the xml document. However, paging is a simplified case compared to supporting any reference to any XML document As a result, supporting paging for large datasets does not introduce the same UI complexity that support for resolving any URL.

4.1.1 Example: Paging of Defect data

This example shows how the list of defects shown in Figure 3 could be broken up into multiple pages. In a real world example there would be thousands of defects on each page. This has been simplified to illustrate how paging works.

```
http://10.0.0.1:9080/RESTServiceName/Defects

<DefectList
    href=http://10.0.0.1:9080/RESTServiceName/Defects?StartId=DEFEC
    T03 TotalPages=2 Page=1 rel=next>
    <Defect>
        <id>DEFECT01</id>
        <Headline>spelling error in login screen</Headline>
        <State>Opened</State>
        <Severity>3-Average</Severity>
    </Defect>
    <Defect>
        <id>DEFECT02</id>
        <Headline>sales tax incorrect if item deleted from
            purchase</Headline>
        <State>Resolved</State>
        <Severity>1-Critical</Severity>
    </Defect>
</DefectList>
```

Figure 7 - Page 1 of defect data

The content shown in Figure 7 is the first page of data returned by the REST service. The URL used to get the page has no specific paging information. In fact, it is the same URL shown in section 3.1. It is up to logic in the REST service to determine when to page data and when not to. In this example, a setting could have been changed on the configuration of the REST service to cause it to page the data. The key point is that paging is opaque to the caller of a reportable REST service. The caller simply recognizes that there are additional pages and continues to get the URL for the next page.

By examining the URL arguments on the root DefectList element in Figure 7, the caller can tell that there are a total of two pages, that this document is currently the first page and that the href value is the URL to the next page. Note that the TotalPages and Page URL arguments are optional. They do not have to be generated by the REST service. In this case they are and the caller is able to update a progress bar control using that data.

Note that the URL for the next page has a StartId argument. This argument is specific to the implementation of the REST service and is opaque to the caller. The REST service should return whatever implementation data is required to efficiently load the next page of data. Reportable REST services do not require any specific URL arguments for this URL and does not restrict it in any way.

The next step for the caller is to HTTP GET the URL for the next page using the href attribute of the root element from page 1. That data is shown in Figure 8.

```
http://10.0.0.1:9080/RESTServiceName/Defects?StartId=DEFECT03

<DefectList TotalPages=2 Page=2>
    <Defect>
        <id>DEFECT03</id>
        <Headline>cancel sale doesn't correctly repaint
            screen</Headline>
        <State>Resolved</State>
        <Severity>3-Average</Severity>
    </Defect>
</DefectList>
```

Figure 8 - Page 2 of defect data

The second page of data does not have an href attribute on the root DefectList element. This is because it is the last page of data. If there had been more than two pages returned, each page would have included the URL to the next page until the last page was returned. The last page never has an href attribute on the root element. This is how callers know that the page sequence has ended. Remember that the TotalPages and Page attributes are optional and can not be relied on for the control flow of paging by the client.

4.2 Support delta ETL load - only extract modified data since last ETL

When executing a delta load, only resources modified since the specified date and time should be returned by the REST URL. It is not required to support delta load, and REST services that do are not required to support it for every resource URL. A REST service may implement delta load for a select set of resource URLs. The goal is to implement this capability on URLs that return large volumes of data. This allows bulk data transfers between systems to include only the data that has changed since the last transfer.

Delta loads are specified by including the ModifiedSince argument in a URL. The value is a date that conforms to a set of specific date formats. These are the Java format specifiers that define the legal values accepted for the ModifiedSince argument. Each expects a different level of precision.

```
yyyy-MM-dd'T'HH:mm:ss z
yyyy-MM-dd'T'HH:mm z
yyyy-MM-dd
```

There are two supplemental formats accepted as well that expects a space instead of a 'T' between the date and time.

```
yyyy-MM-dd HH:mm:ss z
yyyy-MM-dd HH:mm z
```

If a URL does not support modified since, the argument should be ignored. The required behavior for a Reportable REST URL that does not support delta load is to return all of the data, ignoring the argument completely. It is also acceptable to exclude some, but not all, of the data that is older than the modification date. The goal here is for the Reportable REST service to do the best it can to reduce the data volume. It is not always possible to do this in all cases, either because the modification information does not exist, or it is too inefficient to exclude it.

A REST service should never return an error if it does not support the argument. There is no mechanism to describe whether or not a URL supports delta loading. Consequently, reporting and metrics solutions could pass a ModifiedSince argument to a URL, even if the modified since capability is not supported.

4.3 Field selection capability to limit the volume of bulk data transferred across the network

The field selection capability is defined as a URL argument named "fields" that supports a sub-set of the XPath 2.0 specification. This is an expression that affects the content included by the XML results. It is different than the filtering capability, defined in section 0, because it is not conditional. Either all XML elements matching the expression are included or none are.

Xpath allows selection of specific nodes in an XML document. In a Reportable REST Service, the xpath argument determines what data should be returned from the product data source. The entire set of all possible data that could be returned is defined by the XML schema described in

section 3.2. In order to support field selection, a Reportable REST service must also support schema generation.

Since XPath 2.0 is an extensive specification, a subset of XPath 2.0 will be supported by a Reportable REST Service. The following table defines the sub-set of Xpath supported.

XPath 2.0 Specification Section	**Supported Functionality**	**Unsupported Functionality**
3.2 Path Expressions	Only the ability to specify the next sub-element by name is supported. For example: Project/Tasks/Task This example would include all Task elements in the returned XML document. Wildcards, as described in 3.2.1.2 are supported, allowing expressions like: Project/Tasks/* Project/*/* */*/* Attribute and child selection as defined in section 3.2.1 Steps is supported. Attributes are selected using either attribute::attrname or @attrName. Child elements are selected by default with no qualifier or using child::elementName. Attribute wildcards are also supported using either attribute::* or @*.	Axes are not supported, like descendant, parent or ancestor. 3.2.1.2 Node tests 3.2.2 Predicates (For example, `Project/Requirements/PRRequirement` [1] would not be supported) 3.2.3 Unabbreviated Syntax 3.2.4 Abbreviated Syntax

Figure 9 - Supported XPath functionality

The following examples demonstrate how the supported XPath functionality allows the caller to select what data will be returned by the reportable REST service.

4.3.1 Fields Argument Examples for field selection

The following examples demonstrate why the particular sub-set shown in Figure 9 is required. They are all examples from the RequisitePro Reportable REST service. The examples use the Learning Project, which defines a PR Requirement type. All of these examples use the following URL:

http://server:port/DataServices/RequisitePro/Learning+Project+-+Traditional/Requirements/PR

The schema for this URL, generated by adding "?metadata=schema", includes the complex type for the PRRequirement. Excerpts from the schema are shown below:

```
<xs:complexType name="PRRequirement">
   <xs:sequence>
      […]
      <xs:element name="FullTag" type="xs:string"/>
      <xs:element name="Text" type="xs:string"/>
      […]
      <xs:element name="Priority" type="xs:string"/>
      <xs:element name="Status" type="xs:string"/>
      […]
      <xs:attribute name="href" type="xs:anyURI"/>
```

The FullTag, Text, Priority and Status elements, along with the href attribute, are used in the examples below to select specific content from the REST service.

4.3.1.1 Example:Select FullTag, Priority and Status from Requisite Pro PRRequirements

This example demonstrates how to construct a fields argument to cause the Reportable REST service to return only those fields. The required fields are selected with the following xpath statement:

```
fields=Project/Requirements/PRRequirement/(FullTag|Priority
|Status)
```

This returns the following XML:

```
<Project Verion="1.0.0">
   <Requirements>
      <PRRequirement>
         <FullTag>PR1</FullTag>
         <Priority>Medium</Priority>
         <Status>Incorporated</Status>
      </PRRequirement>
      <PRRequirement>
         <FullTag>PR2</FullTag>
         <Priority>Low</Priority>
         <Status>Approved</Status>
      </PRRequirement>
      <PRRequirement>
         <FullTag>PR3</FullTag>
         <Priority>Medium</Priority>
         <Status>Proposed</Status>
      </PRRequirement>
      […more requirements]
   </Requirements >
</Project>
```

This example shows how individual elements can be selected, and how selectors can be grouped with parenthesis.

4.3.1.2 Example – Select the href attribute

This example adds the generation of the href attribute of each PRRequirement. The href attribute is generated as an attribute in the schema. The added text is shown in bold. The first argument uses the attribute axis from the XPath specification. The second URL uses the abbreviated form, also from the specification.

```
fields=Project/Requirements/PRRequirement/(FullTag|Priority
   |Status|attribute::href)
fields=Project/Requirements/PRRequirement/(FullTag|Priority
   |Status|@href)
```

This returns the following XML, which now includes the href attributes:

```
<Project Verion="1.0.0">
   <Requirements>
      <PRRequirement
         href="http://server:port/DataServices/RequisitePro/Learn
         ing+Project+-+Traditional/Requirements/PR/PR1">
         <FullTag>PR1</FullTag>
         <Priority>Medium</Priority>
         <Status>Incorporated</Status>
      </PRRequirement>
      <PRRequirement
         href="http://server:port/DataServices/RequisitePro/Learn
         ing+Project+-+Traditional/Requirements/PR/PR2">
         <FullTag>PR2</FullTag>
         <Priority>Low</Priority>
         <Status>Approved</Status>
      </PRRequirement>
      <PRRequirement
         href="http://server:port/DataServices/RequisitePro/Learn
         ing+Project+-+Traditional/Requirements/PR/PR3">
         <FullTag>PR3</FullTag>
         <Priority>Medium</Priority>
         <Status>Proposed</Status>
      </PRRequirement>
      […more requirements]
   </Requirements >
</Project>
```

4.3.1.3 Example: Select all attributes using a wild card

You can use the wild card character '*' to select all attributes supported by context element. In this case href is the only attribute. Either the attribute:: axis or the short form are supported.

```
fields=Project/Requirements/PRRequirement/attribute::*
fields=Project/Requirements/PRRequirement/@*
```

This returns the following XML. Since the selectors for FullTag, Priority and Status have been removed, these properties are not generated.

```
<Project Verion="1.0.0">
   <Requirements>
      <PRRequirement
         href="http://server:port/DataServices/RequisitePro/Learn
         ing+Project+-+Traditional/Requirements/PR/PR1"/>
      <PRRequirement
         href="http://server:port/DataServices/RequisitePro/Learn
         ing+Project+-+Traditional/Requirements/PR/PR2"/>
      <PRRequirement
         href="http://server:port/DataServices/RequisitePro/Learn
         ing+Project+-+Traditional/Requirements/PR/PR3"/>
       […more requirements]
   </Requirements >
</Project>
```

This XML document would include any other attributes of the PRRequirement element if they were defined. Since href is the only attribute supported, it is the only one returned.

4.3.1.4 Example: Select all PRRequirement properties

You can use the wild card character '*' to select all elements at a particular level in an XPath expression. In this case, we are selecting all of the elements that are children of PRRequirement.

```
fields=Project/Requirements/PRRequirement/*
```

This returns the following XML:

```
<Project Verion="1.0.0">
   <Requirements>
      <PRRequirement
         href="http://server:port/DataServices/RequisitePro/Learn
         ing+Project+-+Traditional/Requirements/PR/PR1">
         <Name/>
         <FullTag>PR1</FullTag>
         <Priority>Medium</Priority>
         <Status>Incorporated</Status>
         <Text>The QBS system shall, upon user request, display
            detailed customer information</Text>
         <HasParent>false</HasParent>
         <HasChildren>false</HasChildren>
         [… more properites]
      </PRRequirement>
      [… more requirements]
   </Requirements >
</Project>
```

Since the wild card * was used in the xpath statement, all immediate child nodes of PRRequirement were included. Note that this does not select multiple levels of elements. Multiple level selection is shown in the next example.

4.3.1.5 Example: Multiple level wildcard selection

The wildcard character '*' can be used at any location and in any combination with other selectors. The following fields argument selects all PRRequirement children, as shown in the previous example, but also adds the children of the Document element.

```
fields=Project/Requirements/PRRequirement/(*|Document/*)
```

This returns the following XML:

```
<Project Verion="1.0.0">
   <Requirements>
      <PRRequirement
         href="http://server:port/DataServices/RequisitePro/Learn
         ing+Project+-+Traditional/Requirements/PR/PR1">
         <Name/>
         <FullTag>PR1</FullTag>
         <Priority>Medium</Priority>
         <Status>Incorporated</Status>
         <Text>The QBS system shall, upon user request, display
            detailed customer information</Text>
         <HasParent>false</HasParent>
         <HasChildren>false</HasChildren>
         [… more properites]
         <Document>
            <FullPath>C:\Program
               Files\IBM\RationalSDLC\RequisitePro\samples\Learn
               ing_Project-Traditional\qbs product
               requirements.prd</FullPath>
```

```
                <FileDateTime>2004-11-05T14:44:09 EST</FileDateTime>
                <Path>C:\Program
                   Files\IBM\RationalSDLC\RequisitePro\samples\Learn
                   ing_Project-Traditional\</Path>
                <FileName>qbs product requirements</FileName>
                <Extension>prd</Extension>
                <Name>QBS Product Requirements Document</Name>
                <Description>This document contains all high level
                   product requirements for the QBS
                   system</Description>
                <DocumentID>1</DocumentID>
                <RequirementsInDocument/>
                <ReqDocumentRevisions/>
                <ReqDocumentType/>
                <ParentPackage/>
             </Document>
             [… more properites]
          </PRRequirement>
          [… more requirements]
       </Requirements >
    </Project>
```

See the w3 schools tutorial at http://www.w3schools.com/xpath/ and the w3c xpath 2.0 specification at http://www.w3.org/TR/xpath20/ for further information about xpath syntax.

Filtering capability to select specific resources for data generation

The filter is an expression that affects the content included by the XML results. It allows the conditional inclusion of specific XML nodes based on logical expressions. The syntax is a sub-set of xpath 2.0. The exact syntax supported is defined in this section.

The goal of supporting this filter information is to allow the Reportable REST service implementer to optimize how the requested information is returned from the product data source. This allows reporting and data warehousing solutions to optimize the functional overhead required to return product data from the REST service.

The filter is specified as an XPath filter on the fields argument. Therefore, both the field selection and the filter are specified in one single expression. The required functionality, a sub-set of the XPath 2.0 specification, is defined in the following table.

XPath 2.0 Specification Section	**Supported Functionality**	**Unsupported Functionality**
3.3 Sequence Expressions	3.3.2 Filter Expressions. A sub-set of this functionality is supported, allowing a basic filter to be defined for resources returned. For example, Project/Tasks/Task/[State="Assigned"] This example shows how an operational report could select only the assigned tasks from a project. Literal values can be delimited by double quotes or single quotes. If there are no whitespace	3.3.1 Constructing Sequences 3.3.3 Combining Node Sequences

	characters in the literal, quotes can also be omitted. The following two expressions are also valid. Project/Tasks/Task/[State='Assigned'] Project/Tasks/Task/[State=Assigned]	
3.5 Comparison Expressions	3.5.2 General Comparisons - The general comparison operators are =, !=, <, <=, >, and >=.	3.5.1 Value Comparisons 3.5.3 Node Comparisons
3.6 Logical Expressions	Support for multiple comparisons using “and” and “or”. Grouping expressions using parentheses – ‘(‘ and ‘)’	fn:not

Figure 10 - Supported XPath filtering functionality

4.3.2 Fields Argument Examples for filtering

The following examples demonstrate why the particular sub-set shown in Figure 10 is required. They build on the field selection examples presented in the previous section. All are examples from the RequisitePro REST service, use the Learning Project, and use the following URL:

```
http://server:port/DataServices/RequisitePro/Learning+Project+-
   +Traditional/Requirements/PR
```

4.3.2.1 Example: Filter Requisite Pro PRRequirements using Stability='High'

In this case, a filter expression has been added to the PRRequirement element, selecting only requirements with a Stability equal to 'High'.

```
fields=Project/Requirements/PRRequirement[Stability =
'High']/(FullTag|Priority|Status)
```

This returns the following XML. Since the filter was used, this is the entire content returned by the REST service.

```
<Project Version="1.0.0">
   <Requirements>
      <PRRequirement>
         <FullTag>PR1</FullTag>
         <Priority>Medium</Priority>
         <Status>Incorporated</Status>
      </PRRequirement>
      <PRRequirement>
         <FullTag>PR5</FullTag>
         <Priority>Medium</Priority>
         <Status>Approved</Status>
      </PRRequirement>
      <PRRequirement>
         <FullTag>PR7</FullTag>
         <Priority>Medium</Priority>
         <Status>Incorporated</Status>
```

```
            </PRRequirement>
            <PRRequirement>
                <FullTag>PR11</FullTag>
                <Priority>Medium</Priority>
                <Status>Approved</Status>
            </PRRequirement>
            <PRRequirement>
                <FullTag>PR13</FullTag>
                <Priority>High</Priority>
                <Status>Approved</Status>
            </PRRequirement>
        </Requirements>
    </Project>
```

Note that Stability is not returned with the data, even though it is part of the filter. This is because the XPath statement only selects FullTag, Priority and Status. Stability could be included by adding it to the selectors, as shown below:

```
fields=Project/Requirements/PRRequirement[Stability =
    'High']/(Stability|FullTag|Priority|Status)
```

This would cause the Stability element to be included in the results, as shown in the excerpt below.

```
<PRRequirement>
    <Stability>High</Stability >
    <FullTag>PR1</FullTag>
    <Priority>Medium</Priority>
    <Status>Incorporated</Status>
</PRRequirement>
```

4.3.2.2 Example: Multi-level filtering

Filters may be specified at multiple levels. This filter selects all the same requirements as in the previous example, but adds an additional filter selecting all TracesTo relationships that are suspect.

```
fields=Project/Requirements/PRRequirement[Stability =
    'High']/(FullTag|Priority|Status|TracesTo/Relationship[Suspect=
    'true']/*)
```

This is an example of a PRRequirement returned using the fields argument. The only relationships included are ones with a Suspect value of true.

```
<PRRequirement>
    <FullTag>PR1</FullTag>
    <Priority>Medium</Priority>
    <Status>Incorporated</Status>
    <TracesTo>
        <Relationship>
            <Suspect>true</Suspect>
            <RelationshipType>Traceability</RelationshipType>
            <Direction>TracesTo</Direction>
            <RelationshipID>{10C2D0CE-84CF-4C80-9166-
                E5A849FC821B}16{10C2D0CE-84CF-4C80-9166-
                E5A849FC821B}1</RelationshipID>
            <RelatedRequirement/>
        </Relationship>
        [… More matching relationships]
```

```
    </TracesTo>
</PRRequirement>
```

This is the excerpt of the XML schema for this URL that defines the TracesTo relationship:

```
<xs:element maxOccurs="1" name="TracesTo" minOccurs="1">
    <xs:complexType>
        <xs:sequence>
            <xs:element maxOccurs="unbounded" name="Relationship"
                type="Relationship" minOccurs="0"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
```

4.3.2.3 Example: Using multi-level paths in the filter

The filtering examples shown up to this point show how filters can be used to limit the XML elements included by the child elements of the element the filter is included on. Up to this point, all of the fields used in the filter have been values of the XML elements directly contained by the filter element. However, you can also use XPath syntax to reach deeper into the child element hierarchy to select values. There is one rule that determines if this is legal. There must be one and only one value for the filter to evaluate. Otherwise, the expression would be undefined.

For example, the PRRequirement has a related document. This is defined in the schema as a xs:element with a maxOccurs of 1 and a minOccurs of 0.

```
<xs:element maxOccurs="1" name="Document" type="ReqDocument"
    minOccurs="0"/>
```

Since there can never be more than one related document, it is possible to evaluate the expression. In cases where there is no related document, there is no match.

This fields expression will execute the filter:

```
fields=Project/Requirements/PRRequirement[Document/Extension=prd]/
    (*|Document/*)
```

Other supported functionality:

Literals: Literal values can be specified using either double quotes single quotes. Either of these expressions is valid:

```
Project/Tasks/Task[State="Assigned"]
Project/Tasks/Task[State='Assigned']
```

This capability is very useful for reporting systems that select a sub-set of the content returned by the REST service in a report. Support for this requirement allows the reporting system to pass a description of the data it needs to a Reportable REST service. This allows the REST service to do significantly less data loading and return the data much more efficiently.

4.3.2.4 Example: Using XML attributes in a filter

The syntax for xpath filtering also supports using attribute values. If the RequisitePro REST service defined Stability as an attribute instead of as a child element, the syntax compared to example 4.3.2.1 would be very similar. The differences are shown in bold.

```
fields=Project/Requirements/PRRequirement[attribute::Stabil
   ity = 'High']/(FullTag|Priority|Status)
fields=Project/Requirements/PRRequirement[@Stability =
   'High']/(FullTag|Priority|Status)
```

Both forms of the attribute axis are supported, "attribute::" and "@". The XML results would be identical, since Stability is not included in the results.

5 Supplemental Requirements

The contents of this section are recommendations for resolving problems that each reportable REST service is likely to encounter. The benefit of each is explained in each section below.

5.1 Authentication

There are three common forms of authentication used by REST services. The preferred form is forms based post because it supports Unicode characters and is the most secure.

Forms based post: The best method is to use HTTP POST to send an html form to a REST service containing the username and password. The following HTML is an example of a form that will post authentication information to a REST service. When you open the HTML in a browser, fill in the form and submit, it sends the form containing the authentication information.

```
<html>
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-
    8">

<script language="javascript">
function submitForm()
{
  var form = document.getElementById("form");
  var url = document.getElementById("url");
  form.action = url.value;
  form.submit();
}
</script>

</head>
<body>
  <form id="form" method="post">
    <table>
      <tr>
        <td>URL</td>
        <td>
          <input type="text" id="url"
   value="http://localhost:8080/restservice/resource">
        </td>
      </tr>
      <tr>
        <td>Username</td>
        <td>
          <input type="text" name="username">
        </td>
      </tr>
      <tr>
        <td>Password</td>
        <td>
```

```
            <input type="password" name="password">
          </td>
        </tr>
        <tr>
          <td>
            <input type="button" value="Submit"
    onClick="submitForm()">
          </td>
        </tr>
      </table>
    </form>
  </body>
  </html>
```

Basic Authentication (optional): Support of Basic Authentication will simplify testing a reportable REST service using a browser. The data service reads the authorization HTTP header to process basic authentication credentials. However, since the basic authentication standard expects the credentials to be base64 encoded, any UTF-16 characters get corrupted.

URL arguments: This method uses the username and password URL arguments for authentication. Use of this form of authentication is not recommended. If a REST service implements this type of authentication, care should be taken in an implementation not to write the value of supplied passwords to logs.

Reportable REST services that require authentication should support one or more of these techniques.

5.2 Handling of illegal XML names

Data exposed through REST interfaces is not always designed for conversion to XML. As a result, the product data used to define element names may contain illegal XML element characters. The REST service needs to apply renaming logic to convert names into valid XML. In the event that the true name does not match the XML element name, annotations should be added to the XML Schema that is used to determine the actual name of the field. The annotations are only added if the element name contains illegal characters.

This is an example of an element declaration with a corrected label annotation.

```
<xs:element type="xs:integer" name="Weight_lbs_">
   <xs:annotation>
      <xs:appinfo>
         <label>Weight (lbs)</label>
      </xs:appinfo>
   </xs:annotation>
</xs:element>
```

Figure 11. Application annotations used to handle illegal XML names

By providing the original name in the metadata schema, reporting and data warehousing solutions have the data needed to correctly display names that are not legal XML values.

5.3 Support for Locale specific names

Reportable REST services designed for use by multiple locales need a technique to return localized names, such as the name of a property or a resource. It is not desirable to rename XML elements for different locales because this affects the interoperability of the XML between clients using those locales. Instead, the strategy is to use application specific annotations in the XML schema. In fact, the same annotations are used as in section 5.2. The only difference is that the label is the name is in the locale of the requestor of the XML schema.

It is important to return names in the locale of the requester rather than using the locale of the server. This allows reporting and data warehousing solutions to get names in the language settings of the user's computer rather than the language settings of the server. Of course, it also requires installing the required language translations for the REST service on the server. The REST service should also default to the language settings of the server if the requested language is not installed.

The XML schema is always the same regardless of either server or requester locale. The difference is the values returned in the annotations. Figure 12 shows what the annotation might contain for French locales for the "Weight (lbs)" property.

```
<xs:element type="xs:integer" name="Weight_lbs_">
   <xs:annotation>
      <xs:appinfo>
         <label>Poids (lbs)</label>
      </xs:appinfo>
   </xs:annotation>
</xs:element>
```

Figure 12. Returning locale specific names in the XML schema

Support for this requirement provides a mechanism to return locale specific names in the metadata schema without affecting the XML data generated by the REST service.

5.4 Date Formats

Standardization of date formats retuned in XML data makes the processing of the data simpler. In many cases it is impossible to distinguish between two different date formats using the string representation of the dates only. For example, consider the following date formats.

MM-DD-YY

YY-MM-DD

What date value is 03-04-05? Since it could be either of the date formats, this could either be March 4, 2005 or April 5, 2003. Unless the caller has specific knowledge of what format the date is returned in, it is impossible to parse dates returned by the service. Therefore, the task of interpreting date values in reporting and data warehousing solutions is simpler if the possible date formats are unique and standard across all services.

All dates are formatted in accordance with the ISO 8601:2004 standard. The formats returned are:

YYYY-MM-DD

YYYY-MM-DDThh:mm z

YYYY-MM-DDThh:mm:ss z

Where:

YYYY – The year in the Gregorian calendar.

MM – The month of the year between 01 (January) and 12 (December).

DD **–** The day of the month between 01 and 31.

hh – The number of complete hours that have passed since midnight, between 00 and 23.

mm – The number of complete minutes since the last hour, between 00 and 59.

ss – The number of complete seconds since the start of the last minute, between 00 and 59.

Note: The java format strings accepted by the java.text.SimpleDateFormat class are:

```
"yyyy-MM-dd"
"yyyy-MM-dd'T'HH:mm z"
"yyyy-MM-dd'T'HH:mm:ss z"
```